CS414 SP 2007 Assignment 3

Due Mar. 5 at 11:59pm Submit your assignment using CMS

1. A dental clinic consists of a waiting room with N chairs and a treatment room. If there are no patient to be treated, the dentist plays solitaire on his computer (which can be considered as a sleeping state). If a patient enters the dental clinic, the dentist will stop playing and start treating the patient in the treatment room(think of this as the dentist being waken from his sleeping state). If the patient comes in and dentist is busy (treating another patient in the treatment room), but there are available chairs in the waiting room, the patient sits in one of the chairs and waits. Otherwise, if all chairs in the waiting room are occupied, then the patient leaves the dental clinic. Write a program to coordinate the dentist and patients using monitor.

```
void dentist_func()
{
    while (true) {
          mon.dentist_wait();
          treat_patient;
    }
}

void patient_func()
{
    if (mon.patient_wait())
          receive_treatment;
    else
          return;
}

monitor mon
{
    condition dentist, patient;
    int num_patient = 0;

    void dentist_wait() {
      if (num_patient == 0)
          patient.wait()
      num_patient--;
      dentist.signal();
    }

    bool patient_wait() {
      if (num_patient == N)
          return false;
      else {
          num_patient++;
          if (num_patient == 1)
             patient.signal()
          dentist.wait()
          return true;
      }
    }
}
```

2.
(a) Consider the following three scenarios. One of them is best described as deadlock, one as livelock and one as starvation. Explain which is which.

(i) Philosophers A and B want to pick up a chopstick. They try to be nice by waiting for one second if the other one is about to take the chopstick. They both try to pick up the chopstick, wait for a second, try again, wait again, ad infinitum.

(ii) Process A is waiting for the result of a computation performed by process B. However, A has higher priority than B, and so the OS prefers A and refuses to give CPU time to B. Therefore, both A and B are stuck.

(iii) Processes A and B communicate through a buffer. Process A is waiting for the buffer to be more full before it reads it. Process B is waiting for the buffer to be less full before it writes more data to it. Therefore, both A and B are stuck.

Answer: (i) livelock (ii) starvation (iii) deadlock

(b) A system has 4 processes and 5 allocatable resource. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
Process A	1 0 2 1 1	1 1 2 1 2	0 0 *x* 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of *x* for which this a safe state.

Answer: If x is 0, we have a deadlock immediately. If x is 1, process D can run to completion. When it is finished, the available vector is 1 1 2 2 1. Now A can run to complete, the available vector then becomes 2 1 4 3 2. Then C can run and finish, return the available vector as 3 2 4 4 2. Then B can run to complete. Safe sequence D A C B.

3. CPUs all have on-chip caches for instructions and data. One design is to have caches *physically indexed and physically tagged*, which means both the cache index and tag are physical addresses. Alternatively, the CPU can index the cache and uses tags with virtual address (*virtually index and virtually tagged*) (there are other two type of designs which we omit for the sake of this question)

(a) Discuss the advantage having *virtually index and virtually tagged* cache over *physically indexed and physically tagged.*

Answer: The motivation to use virtual addresses is speed: a virtually-indexed, virtually-tagged cache cuts the MMU entirely out of the cache lookup process.

(b) You are asked to consult on the design of Lunix again. (Recall from hw1, that Lunix is a simple kernel designed to support multiprogramming on a single-processor machine). The designers tell you that they have implemented and tested (successfully) the Lunix's virtual memory management component. However after they added the support for *memory-mapped files,* one test case failed. The relevant C code snippet of the test case looks like this:

```
fd = open("TestFile", O_RDWR);
ptr1 = (int*)mmap(some_addr, some_length, PROT_WRITE, MAP_SHARED, fd, some_offset);
ptr2 = (int*)mmap(some_addr, some_length, PROT_WRITE, MAP_SHARED, fd, some_offset);
*ptr1 = 1;
*ptr2 = (*ptr2)+1;
close(fd);
```

Essentially what the code does is that it opens a file, memory mapped the same segment of the file twice, and modify the mapped memory (hence the file). The designers find out that incorrect value gets written to the file "TestFile" if the code is running on system with *virtually index and virtually tagged* caches; however, on a *physically indexed and physically tagged* machine, the outcome is correct. What do you think is the problem? How would you change Lunix to address this problem.

Answer: This is the problem known as *virtual aliases.* Multiple virtual addresses (in this case ptr1 and ptr2) mapping to a single physical address can be a problem. Here, both *ptr1 and *ptr2 are in the cache, their modification will not be visible to each other and potentially overwrite each other when updating the actual physical memory. One thing can be done in the kernel is whenever data in the shared region is altered, flush the cache and invalidate the cache entries. Though, modern processors are likely to address the issue in hardware.

4. In a 32-bit machine we subdivide the virtual address into 4 segments as follows:

8-bit	8-bit	6-bit	10-bit

We use a 3-level page table, such that the first 8 bits are for the first level and so on. In the following questions, sizes are in bytes.

Note: The problem asked for the actual allocated page table size, not the number of valid PTE. The solution below gives both answers.

(a) What is the page size in such a system?

2^10 = 1K

(b) What is the size of the page tables for a process that has 256K of memory starting at address 0? Assume each page-table entry is 4 bytes.

# valid PTE : (1 + 4 + 256)
Actual page table size: (256 + 256 + 4 * 64) * 4Bytes = 3072 Bytes

(c) Also assume that each page-table entry is 4 bytes. What is the size of the page tables for a process that has a code segment of 48KB starting at address 0x01000000, a data segment of 600KB starting at address 0x80000000 and a stack segment of 64KB starting at address 0xF0000000 and growing upward (address of top of stack increases)?

# valid PTE: (1 + 1 + 48) + (1 + 10 + 600) + (1 + 1 + 64)
Actual page table size: ((256 + 256 + 64) + (256 + 64*10) + (256 + 64)) * 4Bytes = 7168 Bytes

5. Suppose that a large file is organized as a binary tree of nodes and that you are using a procedure that searches the tree. Each node is the size of one page. Every operation on the tree involves a search that starts at the root. Suppose that this tree is the only paged data segment in your program.

(a) Give a brief definition of the term "working set"

WS is the stuff that's used extensively and repeatedly

(b) Which pages of the tree are likely to belong to the working set?

The upper layers of the tree are the WS

(c) Suppose that the tree has depth 500, that most searches descend all the way to a leaf node, and that the operating system can assign only 250 page frames of physical memory to your program. What problem might arise if the LRU page replacement algorithms is used to manage this memory?

Top layers of the tree will be thrown out during traversal, so they need to be reloaded each time. Ends up fault on every page.

(d) Suppose the conditions are as in part 3(c), except that now 750 pages of the physical memory are available for your program. Now would LRU work significantly better as a page-replacement policy? Explain briefly

LRU would work better in this scenario. Top layers of the tree are likely to be re-touched at the start of each search, hence become "recent" and less likely to be thrown out. However, it also depends on the search's access pattern. If the paths of consecutive searches tend to deviate very early, then the worst case of faulting on almost every page can still happen.

(e) Suppose that you had the option of implementing your own page-replacement policy. What policy would you propose? would it significantly out-perform LRU?

Conceptually we could have an algorithm that picks the victim based on tree depth. Each page records its depth, and the page with biggest depth (closer to the leaf) is replaced. This way, the upper layer of the tree (working set) are guaranteed to stay in the memory. Again whether it will "significantly" out-perform LRU depends on the access pattern. For example, given we have 250 page in the memory, and repeatedly search for the same node, then this algorithm will be twice better than LRU.

It may seems that MRU also works. Indeed if the access pattern is the same as the one in above example, MRU will perform comparably. However, if the search path deviates very early, MRU tends to perform poorly (Imagine a search starts with the left-most tree, followed by a sequence of right-most tree searches, then MRU will fault on every page for the right-most tree searches)
One generalized policy that resembles the above algorithm is LFU (possibly with aging).

6.
(a) Why are shared libraries highly desirable?

Shared libraries mean less space (memory and disk) wasted on redundant code. Allow library updates to fix bugs.

(b) Why are shared libraries sometimes undesirable?

The runtime performance costs of dynamic linking are substantial compared to those of static linking, part of the linking process is moved to execution time (dynamically linked symbol need to be resolved). Library version control are not sophisticated, sometimes library upgrades can break applications.

(c) For the following statement indicate True or False, briefly explain: One problem with heavy use of threads for parallelism and dynamically linked files for sharing is that an application can end up with a large, fragmented, and sparse virtual address space.

The sentence could be True or False. Yes, the use of threads and dynamically linked file will end up with a fragmented, sparse virtual address space. For each dynamically linked file, kernel need to setup at least two segments (code and data). Each threads need its own stack. All these segments will not be arranged consecutively. Hence, the virtual address space will end up being fragmented. However, having a fragmented and sparse virtual address space is not necessarily a problem. See part (d)

(d) For the following statement indicate True or False, briefly explain: Large fragmented and sparse virtual address spaces diminish performance by reducing spatial locality and consequently lowering cache hit ratios.

False. First having the address space being fragmented and sparse have no effect on the temporal locality. Whether or not it has effect on spatial locality depends how the fragments are setup. If the fragmentations are caused by the reasons describe in part(c), then spatial locality is not affected. Because, spatial locality generally exist within a logical segment, and in the case with (c), the logical segments remain whole (undivided).